# Higher Order Functions: Team Worksheet

## Objectives

After completing this worksheet, you should be able to:

* Write code that passes functions as arguments.
* Write functions that return other functions.
* Relate execution context and scope to closure.
* Write code that utilizes closure.
* Define and identify lexical this regarding arrow functions.

## Model A | Higher Order Functions (10 minutes)

As we saw in the previous worksheet, functions are first-class in JavaScript, which means they are treated like any other expression.

**Because wrapper takes a function as an argument, it is considered a higher-order function.**

| const wrapper = (fn) => fn(); const hi = () => console.log("hello world"); wrapper(hi); *// What will this do?* |
| --- |

1. What syntax do we use to call a function?  
   We put arguments in parentheses after the function name.
2. What does hi do when called?  
   It will log “hello world”
3. What is the *parameter* of wrapper?  
   fn
4. What is being passed as an *argument* to wrapper?  
   The function named hi
5. What gets logged when this code is run?  
   “hello world”
6. What does wrapper do? Try to be as specific as possible.  
   Wrapper takes a function in as a parameter, and returns the return value of calling that function without parameters.

*Note*: You have actually seen higher-order functions before,callback functions!

## Model B | Returning Functions (25 minutes)

**Functions that return functions are also considered higher-order functions.**

| const adder = (x) => (y) => x + y;  *// Here is the same function written without arrow syntax* function otherAdder(x) {  return function (y) {  return x + y;  }; }  const add2 = adder(2); const add4 = adder(4);  const num1 = add2(5); *// What will num1 be?* const num2 = add4(5); *// What will num2 be?* |
| --- |

1. What parameter does adder take?  
   adder takes a single number x
2. What does adder return?  
   adder returns a function that takes a single number y and returns x + y
3. What is the type of add2?  
   add2 is a function with type number -> number
4. What number is num2 initialized to?  
   num2 is initialized to 9
5. Use adder to create a function that adds 8 to its input.  
   const add8 = adder(8);
6. Write a multiplier function that takes some number x and returns a function that takes some number y and returns the product of x and y.

| *// Function syntax* function multiplier(x) {  return function (y) {  return x \* y;  }; }  *// Arrow function syntax* const multiplier = (x) => (y) => x \* y; |
| --- |

Let's take a look at a *slightly* more realistic example using factory functions! In this example, `createRunner` is a **higher order function** that returns a factory function.

| const createRunner = (speed) => (name) => {  return {  name,  speed,  location: 0,  move: function () {  this.location += speed;  },  }; };  const createFastRunner = createRunner(7); const createSlowRunner = createRunner(3);  const foo = createFastRunner("FOO"); const bar = createSlowRunner("BAR"); |
| --- |

7. What is the speed of foo?  
foo has speed of 7

8. What is the speed of bar?  
bar has speed of 3

9. What location do all Runners start at?  
All Runners start at location 0.

10. What does the `move` method do?  
Move increases a Runner’s location by its speed.

11. Is `createFastRunner` a factory function? What about `createSlowRunner`?  
They are both factory functions because they explicitly build and return an object.

12. Write code to:

1. Initialize a variable `baz` to a fast runner with the name "BAZ"  
 const baz = createFastRunner(“BAZ”);

2. Initialize a variable `thud` to a slow runner with the name "thud"  
 const thud = createSlowRunner(“thud”);

13. Use `createRunner` to write a new factory function that returns Runners with a speed of 15.  
 const createQuickRunner = createRunner(15);

14. (*Extension*) How might this same technique be achieved using constructor functions or class syntax?  
We can create FastRunner or SlowRunner classes that extend a Runner class.

## Model C | Closure (20 minutes)

Let us take another look at the code from the previous model.

| const adder = (x) => (y) => x + y;  *// Here is the same function written without arrow syntax* function otherAdder(x) {  return function (y) {  return x + y;  }; }  const add2 = adder(2); const add4 = adder(4);  const num1 = add2(5); *// What will x be?* const num2 = add4(5); *// What will y be?* |
| --- |

1. When adder(2) is called, what is the value of x?  
   2
2. What is the value of x when adder(4) is called?  
   4

We have encountered different types of *scope* in the past, such as global or local. This has to do with something called **execution context**, which you can think of as a snapshot of all the variables that JavaScript can access at a point in time.

All JavaScript programs start in the **global execution context**.

When a function is invoked, it is put on the *call stack* and a **function execution context** is created. That is why variables that are declared inside a function is unable to be accessed outside of the function. JavaScript leaves the function execution context once the function is taken off the call stack.

1. In the global execution context, what does num1 point to?  
   Points to the value 7, which is the return value of calling add2(5)
2. When adder(2) is called, what is the value of x in the function execution context?  
   2
3. When adder(4) is called, what is the value of x in the function execution context?  
   4

Despite having the same function body, add2 and add4 are able to "remember" different values of x. That is due to the fact that they store different **lexical environments**.

1. In add2's lexical environment, x is 2. What is x in add4's lexical environment?  
   4

A **closure** is created when a lexical environment is combined with a function that uses that specific lexical environment.

*Note:* There is a lot of vocabulary introduced in this model. While it is important to be aware of these terms, it is not necessary to memorize all of the definitions. When you debug your programs, you might run into errors with scope. That is when you look back at these notes.

## Model D | Lexical this (10 minutes)

While arrow functions are usually identical to their traditional counterparts, several differences do exist. One important difference is that arrow functions **do not** have their own this.

Consider this example:

| const pet = {  species: "dog",  sound: "woof",  shout: () => console.log(this, this.sound),  yell() {  console.log(this, this.sound);  }, };  pet.shout(); pet.yell(); |
| --- |

The Developer should copy and run this code.

1. What does pet.shout() log?  
   {} undefined
2. What does pet.yell() log?  
   {species: “dog”, … etc } woof
3. What is the value of this in pet.shout()?  
   An empty object
4. What is the value of this in pet.yell()?  
   The object stored in the pet variable.

What you are seeing is because of something called **lexical this**; arrow functions bind this to the lexical environment that the function is declared in, which in this example, is the global execution environment.

## Model E | Currying (15 minutes)

**Currying** is another programming technique that utilizes closures and scope.

| const url = (protocol) => (subdomain) => (domain) => (tld) => (endpoint) =>  `${protocol}${subdomain}.${domain}.${tld}/${endpoint}`;  const fullstack = url("https://")("www")("fullstackacademy")("com"); const weatherApi = fullstack("weather"); const stocksApi = fullstack("stocks");  *// "Uncurried" version of `url`* const url2 = (protocol, subdomain, domain, tld, endpoint) =>  `${protocol}${subdomain}.${domain}.${tld}/${endpoint}`; |
| --- |

url is a **curried** function because it is a sequence of functions that each take a single argument. This allows for something called **partial application**, which means that only a portion of the parameters have been binded to values.

1. What is stored in fullstack?  
   A function that takes an endpoint as a parameter and returns the string “<https://www.fullstackacademy.com/>” + that endpoint
2. What is stored in weatherApi?  
   The string “https://www.fullstackacademy.com/weather”
3. What is the type of stocksApi?  
   string
4. In fullstack's lexical environment, what is the value of subdomain?  
   www
5. Use url to create a function where protocol is bound to "http://".  
   const http = url(“http://)
6. What is `url.length`? What is `url2.length`?  
   1 5
   1. Are the lengths the same or different? Why do you think that is?  
      The lengths are different because url is a curried function while url2 is not.
   2. What do you think the `length` property of a function represents?  
      The length property represents the number of parameters the function has.